The ADE Environment

A Macroscopic Object Architecture for Software Componentization and Distributed Computing

David Slik <dslik@uvic.ca>

This paper describes a software runtime environment for the development of distributed component-based software. Called the Asynchronous Distributed Environment, or ADE, it permits rapid software development of distributed applications spanning multiple computers.

The ADE environment is based on software components called "Modules". A Module can be thought of as macroscopic objects that act and run as separate entities. Modules can range in size from a small data structure manager to a user interface handler to a complete server. Except for the ability to send messages to other Modules, each Module is completely isolated of the rest of the software running on the computer. Modules communicate by sending these messages between each other. With this basic communication mechanism, relationships can be defined and full software systems can be built up. The resulting systems are fully asynchronous, utilize preemptive multitasking and support distributed computing across a network.

Modules:

Easily implement state machine servers, or even complete subsystems.

Easily integrate with each other and existing systems.

Can be programmed in any language, both object oriented and non-object oriented.

Inter-operate, regardless of their native language.

Can be built from existing code.

Support scripting and have hooks for visual languages.

Scale from a single system to a multiprocessor to a multicomputer.

Through this environment, this paper addresses portions of the scalability problem in software development. These include scalability of the software and of the development effort required.

1. The Componentization Problem

One of the core problems in software development is how to manage complexity. At the surface, it appears that software development time is the sum of the time required to solve the problems that make up the system. After all, software is built by combining solutions to smaller problems. Unfortunately, this is not the case. Large system development efforts require much more time then that would be required to solve the component problems separately. This disparity has become known as the integration problem. Thus, how software is broken into components and how these components interact is fundamental to the integration problem and consequently has a direct impact on complexity.

Componentization of programs has been with computing for a long time. Program fragments were written separately and reused back when programs were

written on punch cards. While this worked for smaller programs, ad hoc segmentation did not last long as the concept of the subroutine rapidly emerged in programming languages. The concept of the subroutine has been such a powerful concept that it still dominates programming languages today. While some programmers used subroutines just to segment code, others used it to reduce the complexity of the development process. They did this by setting up enforced guidelines and adding the ability to restrict access to local variables within a subroutine. As a result of their success, the subroutine has become a core part of the programming mindset.

Over time, these concepts of isolation became part of the central tenants of structured programming. Combined with guidelines on branching and restrictions on random jumps in execution, structured programming had far reaching effects. But while it had reduced complexity, the development it enabled rapidly nullified any gains. With program sizes and problem domains growing at an ever-increasing rate, software grew in complexity faster then mechanisms could be put in place to manage it.

A few people in the industry and research community saw that subroutines were just a first step. After investigating how people tended to think and program, they proposed that data and associated routines should be treated as objects. These objects would be made the focus of the development process. By structuring these software objects like objects in real life, the concepts of properties and actions would make sense at an intuitive level. When object oriented languages are used, the way that programmers make their software interact are very different then when using subroutines. Subroutines are usually tightly integrated, with the implementation having a direct impact on how the routines are used. With objects, the implementation and interface could be truly separated and less tightly coupled.

In a way, Object Orientation can be thought as a stricter form of structured procedural programming. On a technical side, it is simply a more rigid set of guidelines that data and procedures adhere to. And like structured programming, it has reduced complexity by a limited amount. Software has continued to grow in size and complexity at an exponential rate. With the advent of faster computers, the beginning of universal network connectivity, software has once again reached the point where a new mechanism to reduce complexity is needed.

Software could have continued to be developed using existing techniques if no new stresses were placed on the development community. But like most areas in computing, change is the only constant. The evolution of the Internet into a universal network service was this next large stress. The changes in software development requirements as a result of worldwide networking are similar to those that occurred during the advent of the graphical user interface. As most software was designed for single computer systems, network connectivity rarely meant more then file and print services provided by the operating system. Now it seems that almost every application is including mechanisms for Internet access, serving, collaborative document creation and networking.

This need for an effective mechanisms to build network enabled distributed applications is what has motivated this paper and the associated proof of concept software package.

If structured programming are objects at the instruction level
If Object Orientation is objects at the data level
Then the next step is objects at the problem level

If instruction programming is structure at the CPU level
If advanced programming interfaces (APIs) is structure at the system level
Then the next step is structure at the network level

This trend is already becoming apparent in most development environments. Software architectures are migrating towards problem domain objects (eg, ActiveX, OpenDoc) and network architectures (eg, DCE, DCOM, and CORBA). The growth of the World Wide Web can be taken as an example of the readiness of the industry for these techniques. The Web can be viewed as a problem level programming language based around objects at a network level (Links and files and images). The popularity of Visual Basic, one of the most used programming languages in the world, is also an indication of this need.

software, the barriers to entry in the software marketplace once again be reduced.

1.2 Examples of componentization in the Industry

1.2.1 LabView

LabView is a visual programming language designed for instrumentation, data acquisition and processing. It has innovative and unique mechanisms for creating software components. It represents each action as a separate object and allows collections of objects to become a reusable object. Thus, large applications are easily built up from simple components. As a result of the visual nature and the implicit data flow architecture of the language, it supports automatic compilation into parallel threads and is thus executed efficiently on single and multi-processor systems.

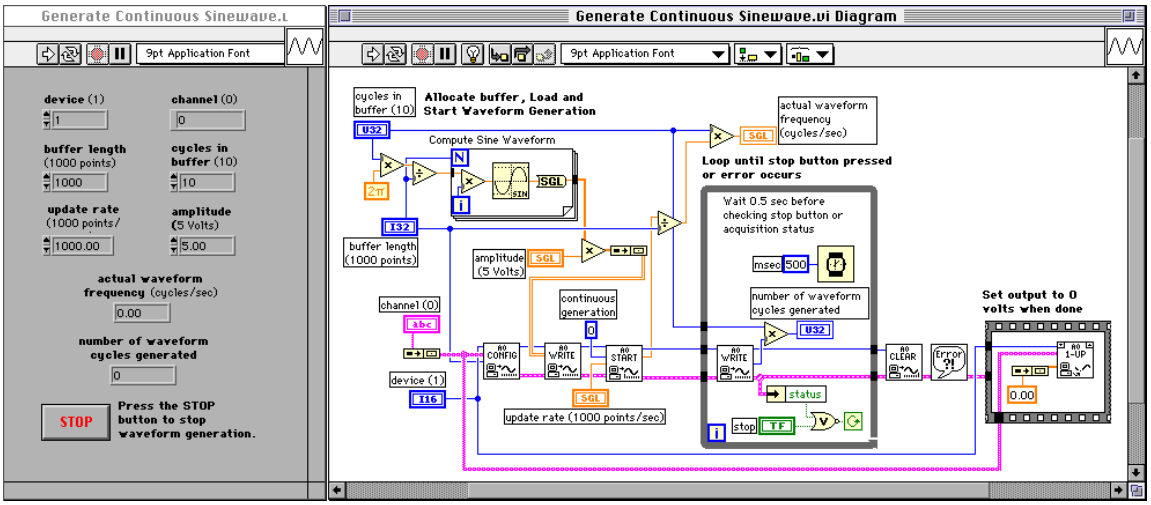

Figure 1.1: *Example LabView interface and diagram*

With the beginning of the acceptance of visual languages and component-based software, this is an exciting time for the industry. As standards merge towards an open and interoperable architecture, software components will be able to work together resulting in software complexity being reduced once again. With the existence of standard components to build

In the above diagram of a LabView program, note the use of coloration to indicate different data types and standard logic and mathematical symbols to represent corresponding operations on data streams.

This approach to software componentization addresses many of the issues raised. As LabView has a consistent runtime and object framework, all objects

work together. Its approach to visual programming allows visual construction of both the logic and the interface. Basic mechanisms for distributed computing are included and allow two or more LabView programs to exchange data over a network.

LabView has been targeted towards a very specific vertical market: data acquisition, processing and display. As it has been focused on this problem space from initial development onward, some of the design decisions make it not suitable for general application development. Despite this, LabView is a development environment that everyone should take a look at to see what is coming.

1.2.2 JavaBeans

Java takes several interesting approaches to software componentization. While Java relies on the principles of object orientation for objects at the language level, JavaBeans provides high level support for reusable user interface components. Given the integrated support for networking and distributed computing, development of object based software for network applications in Java has flourished.

The Java language is based on syntax similar to that of languages such as Ada, C and C++, which came before it. It is text based and built on the principles of object orientation. Distributed computing capabilities currently are focused around remote method invocation. This is a variant of the Remote Procedure Call updated for the newer object oriented architecture.

JavaBeans provide a standard template used to create medium weight reusable components. These components can be integrated directly into Java programs through written code or assembled together using a visual development tool. Almost all JavaBeans are developed with a user interface. Support for Beans without an interface has just been introduced as Enterprise Java Beans. An example of a JavaBean is an editable text box that displays bullets when a user enters a password.

Java has been exciting for the development community. It solves and addresses many concerns developers had with older languages. It is safe and easier to develop in, network aware, platform independent and has a wide range of libraries available. Java is poised to become the next general-purpose text based programming language, if not a complete environment.

1.2.3 OCX and ActiveX controls

ActiveX controls (Formerly OCX and VBX controls) are high level user interface objects, similar in nature to JavaBeans. While ActiveX controls are not platform independent like JavaBeans, they are compiled natively and thus run efficiently. ActiveX controls are based on Microsoft's COM (Common Object Model) object model and the OLE (Object Linking and Embedding) framework. While ActiveX lacks the elegance of design found with JavaBeans, they are widely accepted in the Windows world. They form the enabling component architecture for Visual Basic, Delphi and most other component based programming environments. It is worth noting that these environments have been very successful, and many large libraries of components are available.

From a Macintosh standpoint, ActiveX has not been adopted as a result of several reasons. These include late introduction date, lack of development tools and conflicts with the Macintosh way of programming. Implementation has been limited to Microsoft software and a few third party products.

2. Distributed Computing

The era of single processor computer is almost over. For the last several years most computers have already had additional processors within them. These processors have provided I/O services and specialized

co-processing. Five years from now, all but the simplest computers sold will have multiple central processing units, either within a single chip or as multiple chips. Almost all of the modern microprocessors like the PowerPC and Intel Pentium already have multiple processing elements within them that allow several instructions to be pipelined, or executed simultaneously. An example of this architecture is AltiVec, Motorola's addition of a new vector arithmetic unit to enhance the PowerPC in performing DSP operations.

The difference between these existing mechanisms and multiprocessing in the future is in how the systems are programmed. Currently, pipelining is transparent to the programmer. In systems with multiple processors and shared memory, properly programmed threads are automatically shared across processors. While this trend of independence of the programming methodology and the hardware will continue, the level at which it has to be taken into account by the programmer will rise towards the problem domain. This can already be seen happening, with the advent of the Internet and the resulting client/server software developments.

In the age where connectivity is taken for granted, the physical barriers used to define the limits of a computer system will blur. Multiple processors will become common in computer systems and these systems will be increasingly tied together. As the software and programming mindset changes, Sun Microsystems' slogan, "The Network is the Computer" will become a reality. The power of distributed computing is not speed increases. It is a promise of cost reduction, effective utilization of resources, and resiliency. This efficiency is already being taken advantage around the world where large UNIX systems support hundreds to thousands of users simultaneously. Such a system is much less expensive then dedicated computers for each user. When resources can be effectively utilized, less computing power is wasted, and thus less is required.

To take advantage of this emerging connectivity and resulting computing power, new mechanisms and methods to develop software are needed.

2.2 Examples in the Industry

2.2.1 OSF/DCE

The Open System Foundation's Distributed Computing Environment (DCE) is one of the most mature distributed software frameworks available. Based around conventional procedural languages and Remote Procedure Calls (RPC), DCE provides a variety of layered services that include security, object stores, directories, time services and a complete distributed file system.

DCE implementations run as extensions to existing operating systems and coexist with existing software. Vendors have implemented DCE on most major operating systems, including the Mac OS. As a result of the wide range of services and platforms supported, many distributed applications have been written using DCE.

2.2.2 CORBA

CORBA, or the Common Object Request Broker Architecture, is an industry wide object model designed for distributed computing. Supported by most major software vendors except Microsoft, CORBA is currently the leading architecture for large scale distributed systems. It is maintained as a standard by the Object Management Group (OMG).

CORBA is not based around the concept of the RPC. The OMG has refined and evolved its network communication concepts to move away from process blocking and to permit the extension of the principles of object oriented software to the distributed sphere. By using an Interface Definition Language (IDL), software writ-

ten in different languages and on different systems can understand and thus exchange different types of data and associated structures.

Like DCE, CORBA is implemented as an environment that runs on top of an existing operating system. CORBA also is useable with programs based on Microsoft's Common Object Model and programs written in Java. CORBA has a wide range of services and is beginning to displace DCE in terms of systems developed.

2.2.3 I_2O

I_2O, while not a complete distributed architecture like the previous examples, shows how the concepts of distributed computing are becoming widespread and prevalent in hardware and software design. I_2O is a distributed message based I/O architecture that is designed for high performance computing systems. Although it is intended for use within a computer system, its architecture is similar to many message based communication systems used in distributed architectures.

In I_2O, I/O devices are connected to the computer via a bus that is used as a packet network. Each I/O device has its own processor, called an IOP. The CPUs and IOPs communicate over the bus by sending small messages. In addition to this architecture being designed for allowing CPUs and I/O devices to communication, it is also intended to serve as a communication foundation for clustering and other tightly coupled non shared memory distributed computing applications.

I_2O is a very interesting driver model as it offers the potential to connect off the shelf hardware directly into a message based distributed computing system. As a result of the abstraction provided, it also allows support of many different devices by writing a single class driver.

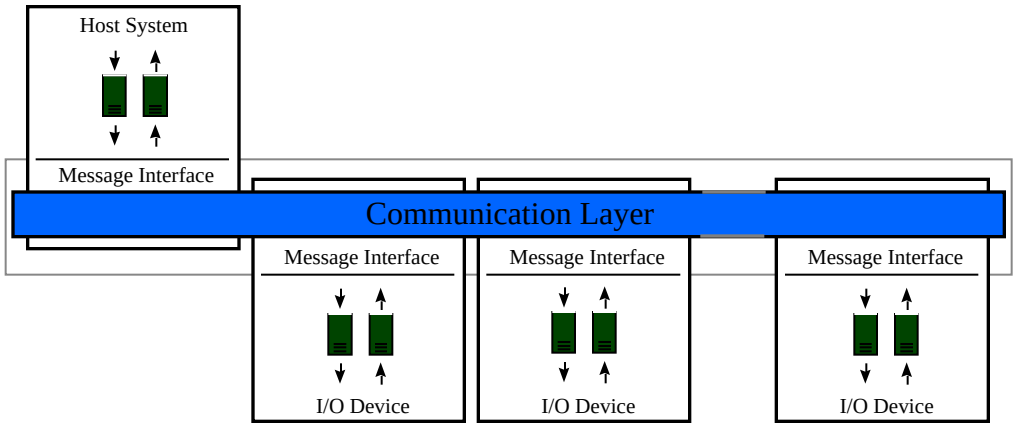

Figure 2.1: *I_2O Architecture Diagram*

3. The ADE Environment

3.1 Disclaimer

The problems found in software development and distributed computing are very complex. The system discussed in this paper, the ADE Environment, is targeted to investigate solutions to a small but important part of the problems discussed above. It should be noted that none of the material and approaches discussed in the paper is new. These ideas, mechanisms and architectures have been extensively discussed in the industry for over thirty years.

What the ADE Environment does provide is an implementation that allows programmers to explore the issues surrounding componentization and distributed computing. It is sufficiently capable to develop commercial grade software in, and is intended to become a platform for future development efforts by the companies involved in its development.

Some of the design tradeoffs in the ADE are disliked. The mindset of

linear procedural programming is still quite strong, and throwing away serialization, message order enforcement, guaranteed delivery and synchronization is seen as extreme, even when at a high level. While many of them can be built into the system by adding modules and libraries, as the system is intended for experimental purposes at this stage, their inclusion at a future date can not be guaranteed. In addition, many of these perceived deficiencies will be removed from view with the development of a planned visual front end.

3.2 Overview

This paper describes an object model and its supporting run time environment used to integrate macroscopic software objects. These macroscopic software objects can then be used to build programs that run on stand-alone and distributed computing systems.

The object model is based around software components called "Modules". A Module can be thought of as a collection of software routines that act and run as separate entities, independent of each other. Modules can range in size from a small data structure manager to a user interface handler to a complete server. A Module can coexist with thousands of other modules, or have its own dedicated computer. These Modules communicate by sending messages to each other. These two capabilities permit complex software systems to be built.

This system has been named the Asynchronous Distributed Environment, or ADE. The messaging format and structure has been named Asynchronous Distributed Messaging, or ADM.

3.2.1 Design Goals

Simplicity
: The system must be simple, both in implementation and architecture. This will permit low runtime overhead in processor cycles and memory. Additionally, it permits the system to be easily developed and debugged.

Clean software model
: It will ensure that software written for the system has a consistent yet flexible architecture and takes advantage of the specialized capabilities provided for it.

Multiple implementation levels
: Allow the system to range from a full operating system to a layer sitting on top of a existing operating system to a gateway permitting access to external systems.

Language independence
: Permit software to be constructed in any common programming language, both object oriented and procedural.

Strong typing and interface binding
: Inputs, outputs and serialized streams are to be strongly typed and meta-information about I/O should permit graphical programming and scripting.

Adoption of emerging hardware trends
: Designed for Hardware I/O using the I_2O specification. Supports native ATM packetization and virtual channel creation. Designed with distributed hardware architectures in mind.

Elimination of synchronization and blocking
: Designed to not require synchronization or blocking. Hardware interfaces do not depend on interrupts.

Loosely coupled architecture using message based communication
: Modules are designed to be completely isolated from each other. They can only communicate to each other via messages, and these messages can be ignored.

Processor-Pool distributed computing
: Users log into a collection of processors. There is no "home" system.

3.2.2 Architecture Advantages

Modules can be easily and quickly developed

Encourages software to be broken into separate components

Encourages designs with higher levels of re-usability

Module size can range from state machines to servers to complete subsystems

Integrates and inter-operate with existing software, libraries and operating systems

Programmable in many object oriented and non-object oriented languages

Modules communication is independent of native language, processor type and node

Existing source code can be used to build Modules

Supports extensions for scripting and visual languages

Scales from a single processor computer to a multiprocessor to a multicomputer

Very low runtime and object overhead

3.2.3 Architecture Disadvantages

Experimental and incomplete implementation

Lack of services and libraries

Software components must interact in an asynchronous fashion

Synchronization at a thread level is not available

3.3 Description

In brief, the ADE system is:

A system comprised of a variable number of preemptive (and cooperative in Mac OS) threads running with portions of the address space mapped into each thread. Threads are logically structured into groups called Modules. Each Module contains process information and has IPC mechanisms that include an input queue and a varying number of ports, which send messages to the input queues of other modules. Module software is structured as an infinite loop state machine that handles incoming events. When combined together, the interacting Modules form a complete software system.

4. Modules and Messages

4.1 Modules

Modules are the processes of the system. A Module is a collection of handlers that interpret messages. Each Handler performs an action on messages received from other Modules. The internal structure and how Modules are implemented is left to the programmer, but as long as they conform to the Module guidelines, they will run and inter-operate.

Example C Module

```
void main(EventPtr theEvent)
{
  unsigned int MessageCount;

  // Constructor
  MessageCount = 0;

  While(!TerminateFlag())
  {
    PullMessage(&theEvent);

    MessageCount = MessageCount + 1;

    Switch(theEvent->EventType)
    {
      case EVENT_ COUNT_QUERY:
      {
        PostReply(theEvent,
          EVENT_COUNT_RESULT,
          MessageCount, 0, 0, 0, 0, 0);
        break;
      }
      case EVENT_COUNT_CLEAR:
      {
        MessageCount = 0;
        break;
      }
    }
  }

  // Destructor would go here
}
```

This module has two handlers that handle two messages. If a message of type EVENT_COUNT_QUERY is sent to the

module, a EVENT_COUNT_RESULT event is returned. Within the result event is the number of messages received by the Module over its lifespan. The second handler is invoked if a message of type EVENT_COUNT_CLEAR is received. This handler resets the counter to zero.

This structure maps directly into the object oriented methodology of programming. If a C++ style object was used to manage the count, the module could be rewritten as follows.

Example C++ Module

```
class Count
{
  private unsigned int InternalCount;

  public Count()
  {
    this.Reset();
  }

  public Incriment()
  {
    InternalCount = InternalCount + 1;
  }

  public unsigned int Get();
  {
    return(IntenralCount);
  }

  public Reset()
  {
    InternalCount = 0;
  }
}

void main(EventPtr theEvent)
}
  theCount = new Count();

  While(!TerminateFlag())
  {
    PullMessage(&theEvent);
    theCount.Incriment();

    Switch(theEvent->EventType)
    {
      case EVENT_ COUNT_QUERY:
      {
        PostReply(theEvent,
          EVENT_COUNT_RESULT,
          theCount.Get(), 0, 0, 0, 0, 0);
        break;
      }
      case EVENT_COUNT_CLEAR:
      {
        theCount.Reset();
        break;
      }
    }
  }
}
```

While this method of coupling events into an object is simplistic, more complicated and efficient mechanisms can be developed. With the addition of introspection, events could automatically invoke the corresponding method in the object. For example, if the objects that the Module was built contained an object called screen that had a method called clear, a Event with a type of EVENT_SCREEN_CLEAR could automatically invoke that method.

4.1.1 Module Theory of Operation

When the Process Server creates modules, new threads and memory spaces are allocated. After the environment is initialized, the main routine of the module is executed. Within the main routine there are three important parts of every module.

The constructor is where variables and storage that is global to the entire module are allocated and initialized. These variables are persistent between messages and are usually used to store global information throughout the life span of the Module.

The Module State Machine is an endless loop that continuously dispatches messages to the corresponding handler. On each cycle around the loop, a flag is checked to see if the module is scheduled to be terminated. If this flag is true, the deconstructor is called. Otherwise, the contents of the loop are executed. First, the PullMessage routine is called. If no messages are pending, the Module is put to sleep. Otherwise, a message is returned. Depending on the contents of the message (usually the EventType field), different handlers are then called.

The deconstructor deallocates memory allocated during the life of the Module and

closes any open ports and streams. It is called just before the module is terminated.

Modules are structured as a state machine that interprets asynchronous events.

The Simplest Module

```
void main(EventPtr theEvent)
}
  While(!TerminateFlag())
  {
    PullMessage(&theEvent);

    Switch(theEvent->EventType)
    {
      case EVENT_BEEP:
      {
        SysBeep(0);
        break;
      }
    }
  }
}
```

This module will cause a Mac OS system to beep when it is sent a message with the EventType field set to the constant EVENT_BEEP.

Actions performed by the module can be blocking or asynchronous. Modules can create their own threads, and process a message while still receiving and acting on other messages. If a module is processing a message, messages received are queued.

4.2 Handlers

A Handler is the programmer-defined block of code that interprets a received message. A Module will consist of one or more handlers. In the earlier section, the sample code of a Module caused the system to beep. The code below is the Handler that was invoked by the EVENT_BEEP message.

EVENT_BEEP Handler

```
case EVENT_BEEP:
{
  SysBeep(0);
break;
}
```

This simple Handler is written directly into the Module switch statement. As the code gets more complex, Handlers are usually moved into a separate routine or object that performs the function when the Handler is invoked. The next example shows how a Handler would be segmented.

EVENT_BEEP Handler - Preferred Handler Separation

```
// From Module.c
case EVENT_BEEP:
{
BeepHandler();
}

// From ModuleHandlers.c
void BeepHandler(void)
{
  SysBeep(0);
}
```

Handlers are usually located in a separate file, named ModuleHandlers.c.

4.2 Messages

Modules are completely isolated form each other. As they are unable to change anything outside their own memory space, modules require a mechanism to allow them to interact. This mechanism is messages. The environment described is completely based on messages. Sent asynchronously from module to module, they form the only recommended method of communication.

Most traditional computing systems with distributed capabilities (NT, UNIX, DCE, etc) focus on the Remote Procedure Call, or RPC. An RPC is a direct extension of standard procedural computing, where the instructions executed are on a different computer. RPCs advantage is that it is based on this traditional style of programming. Its disadvantage is the same. Linear, instruction by instruction style programming is complex and not easily parallelizable.

The message format chosen for this system is the ADM (Asynchronous Distributed Messaging) format that was developed for low cost messaging over ATM networks. ATM offers several advantages, and in many ways is directly suited to the ADE

system. As a connection-based networking architecture, most of the overhead is in establishing the initial connection. As the ADE system handles routing, static connections can be set up between different nodes at startup. This permits ATM drivers to be used efficiently. Packets can be directly passed to these connections without having to use high-level driver functions. ADM packets can also be sent over TCP/IP and ADSP protocols.

4.3.2 ADM Message Format

The ADM message, the mechanism for all message and event based information transfer in the system is based on a 53 byte ATM frame. Of this, five bytes are reserved for use by the ATM delivery protocol, leaving 48 bytes. These bytes are further divided up into twelve 32 bit words, the first half are reserved for system information, and the last half open for programmer use. Data transfer requiring more then 24 bytes worth of data should be accomplished using streams.

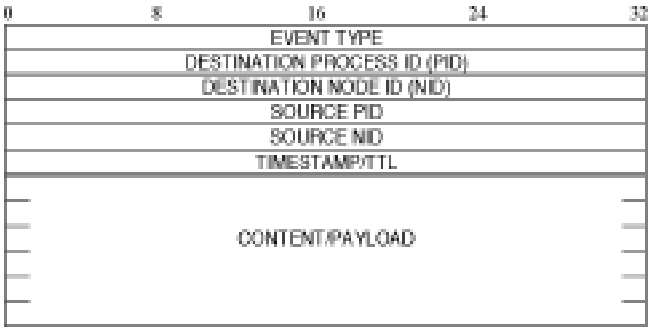

Figure 4.3.2: *Messages in Memory*

All fields of messages are accessible to programs. When messages are sent, some of the fields are overwritten by system information, and thus should not be used.

Message Header

```
//-------------------------------------------------------------
// COPYRIGHT (c) 1997, 1998, David Slik. All Rights Reserved
//-------------------------------------------------------------
// Contents       : ADE Event Header (Public)
//                    (EventServicesLibrary.h)
// Date Created   : July 11, 1997
// Written By     : David Slik
//-------------------------------------------------------------

// STRUCTURES ----------------------------------------------

typedef struct
{
  uint32  EventType;
  uint32  DestinationPID;
  uint32  DestinationNID;
  uint32  SourcePID;
  uint32  SourceNID;
  uint32  Timestamp;
  uint32  Slot1;
  uint32  Slot2;
  uint32  Slot3;
  uint32  Slot4;
  uint32  Slot5;
  uint32  Slot6;
} EventStructure, *EventStructurePtr;
```

4.3.3 ADM Message Fields

EventType

The EventType field contains an identifier that corresponds to the type of event the message is about. The receiving module uses this field to decide what action should be performed as a result of the reception of the message

There are three ranges of identifiers:

Reserved Identifiers	(0 - 4096)	Internal and reserved
System Identifiers	(4096 - 65535)	System modules
User Identifiers	(65545 - 232)	User Modules

EventType Identifiers are registered to prevent conflicts.

Destination Process ID

The Destination PID is the unique identifier that refers to the module that the message is sent to. Each module, when created is assigned an unique identifier. This identifier is stored in the process tree and is used by the dispatcher to route messages to the module.

Destination Node ID

The destination NID is the unique identifier that refers to which node (host) that the message is sent to. Every system that is connected together in the system is assigned a address that is used to route to it. Currently the system topology is a eight dimensional torus.

Source Process ID

The source PID identifies the module that sent the message. This field is auto-

matically set by the internal IPC routines and can not be modified by the module.

Source Node ID

The source NID identifies the system from which the message originated.

Timestamp / TTL

In local messages, the Timestamp / TTL field holds a local timestamp of the system clock when the message was sent. On message destined to remote systems, the Timestamp / TTL holds a Time-To-Live (TTL) value that is used to prevent routing loops.

Payload

The payload is a contiguous set of twenty-four bytes that is used for passing parameters. Each different EventType identifier will define what type of data is stored in the payload fields and how it should be structured.

4.3.4 Message EventType Naming Conventions

As EventType names are usually #defines to registered EventType identifiers. To help make these identifiers easier to use, most programmers add a textual name. Several guidelines exist to help prevent naming conflicts and maintain a consistent naming convention.

All system and server messages begin with the EVENT_ constant. Then comes the object specifier, then comes the action or result. Examples of typical naming include:

EVENT_WINDOW_CLOSE
EVENT_PROCESS_CREATE
EVENT_MEMORY_AMOUNT

To prevent event conflicts, it is recommended that programmers use prefixes for their messages. An example would be:

STI_STATUS_QUERY
DSTI_STATUS_RESPONSE
IRIS_TREND_RESPONSE

It is recommended that a four-letter representation of the company or product name is used.

4.3.5 Message Use Recommendations

As how messages are used are largely dependent on the architecture of the software and the style of the programmer, few restrictions are placed on message use.

The SourceID, SourceNID, TimeStamp / TTL are set by the kernel. Therefore these fields are only useful for events that have been received.

For more information about posting and retrieving messages, see section 5.1.3 for API details.

4.3 Streams

Streams are the data flow programming aspect of the system.

They are one way, read only mechanisms to move bulk data from one module to another. Streams can be connected through modules called filters. These modules function much like filters do in the UNIX operating system. There are two types of streams, active streams, which are data stored in a time addressable form and dormant streams, which are in a space addressable form. An Active stream can be made dormant, and Active streams can be started from any part of a dormant stream.

The Stream Package is currently under development and has not yet been released. The software package that accompanies this document does not include it. When development has progressed to a releasable stage, it will be made available for download at the project site listed at the end of this document.

4.4 Routers and Buses

Routers and Buses form the broadcast and multicast mechanisms for message delivery. They provide message delivery functionality between local and remote Modules.

4.4.1 Local Routing

Local routing is accomplished by a system level call that dispatches messages

to and from Modules running on a specific system or processor. Messages posted using this function call are routed to the queue corresponding to the destination Module ID in a first come first serve basis. Except for when an error condition occurs, the internal dispatcher is completely transparent. If a message is sent to an invalid Module ID or the security permissions are incorrect, then the dispatcher will post an ERROR_UNREACHABLE message to the sending Module.

4.4.2 Remote Routing

When a message is sent to the dispatcher that has a non-local node address, the dispatcher sends the node to a special module that forms a routable bridge over the network connecting the nodes. This bridge will forward the message to the destination node. The routing architecture is based on a fixed topology where each node has up to sixteen connections to its nearest neighbors. When a message is routed, it is sent towards the node, rather then to it.

As the message travels, the Timestamp field becomes a time to live field. As clock synchronization is not used, this field is not worth much between nodes. It should be noted that the lack of clock synchronization is a design decision and can be implemented on top of the system if required.

Each node is defined by a 32 bit node address. This node address is based on adjoining nodes and will not change during operation. The 32 bit word is broken up into eight four bit vertices. These values define the virtual position of the node in a eight dimensional torus or hypermesh.

Listing 4.4.2: *Address of the Seed Node*

Vertices Value	1000 | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 | 1000
Node ID	0x88888888

For example, if a Module on the seed node sends a message to a Module running on node ID 0x88878888, the node will be sent down the lower connection corresponding to the fourth dimension of the hypermesh.

When a message destined to an external node is received, a routing strength vector is calculated. Depending on this vector, the message is routed down one of sixteen virtual channels created over a network to the other nodes. Thus propagation time is directly dependent on how far away the node is. In the previous example, the strength vector would have a strength of one and a direction equal to the 4D Lower connection.

The Remote Routing Package is also currently under development and has not yet been released. It is currently based on the ADSP protocol, and thus runs as a cooperative thread. This is a major performance limitation and also requires the network topology to be manually configured. Consequently, as improvements are made to this package, this document does not include it. When development has progressed to a releasable stage, it will be made available for download at the project site listed at the end of this document.

4.4.3 Buses

A Bus is a special module that takes a single message in and send many messages out. The only bus implemented in this version of the system is a module that maintains a list of modules that are to be notified if a message is sent. Modules can enlist or delist themselves. As buses relay messages, like bridges, they send messages that appear to be from a Module ID other then their own Module ID. This may have to be addressed in a later release for security reasons.

Bus Module Documentation

```
// To join a bus, send a Message with EventType of
// EVENT_BUS_JOIN
// To leave a bus, send a Message with EventType of
// EVENT_BUS_LEAVE

// At this time, Modules cannot check what modules are
// members or delist other Modules.
```

5. The Runtime Architecture

5.1 The Kernel

The ADE (Asynchronous Distributed Environment) Kernel consists of four components, a Hardware Adaptation Layer, a virtual memory manager, a threading manager and a interface library that make the Kernel services available. These services provide the underlying foundation on which a complete distributed computing architecture can be constructed.

The Kernel has two different builds: one that implements all of the components listed above and one that uses services provided by a host operating system. Functions and capabilities available to modules are directly dependent on which build is used. The build that is included with this paper is the second type. It is built using wrappers that use the functionality of the Mac OS.

5.2 Kernel Memory Services

The Kernel Memory Services manages memory pages and their relationship to processes. It is called extensively to swap execution spaces. Each thread has a variable number of pages mapped to it. When the thread context is changed, the pages of the suspended thread are mapped out, and the pages of the revived thread are mapped in. When the Kernel is bootstrapped, the region of address space that is linked to the physical hardware addresses that the memory is stored in is registered and segmented. For hardware driver Modules, hardware addresses can be mapped into the Modules memory space.

5.2.1 Kernel Memory Services

When a Module is first created, the Kernel has automatically mapped in a specific amount of memory that contains the executable code, data and space for execution. When the Module has first been loaded and has consequently has a linear unfragmented memory map. At this time the module has the opportunity to grow or shrink the amount of space available for execution (the stack and heap).

Memory Library Prototypes: (MemoryServicesLibrary.h)

```
// PROTOTYPES ------------------------------------------

// Module Memory expansion and contraction functions
uint32 MemoryGetHeapSize(void);
uint32 MemoryGetHeapFree(void);
uint32 MemoryGetHeapTop(void);

OSErr MemoryGrowHeap(uint32 Amount);
OSErr MemoryShrinkHeap(uint32 Amount);

// Stack manipulation functions
uint32 MemoryGetStackSize(void);
uint32 MemoryGetStackFree(void);
uint32 MemoryGetStackBottom(void);

OSErr MemoryGrowStack(uint32 Amount);
OSErr MemoryShrinkStack(uint32 Amount);
```

Modules can also access additional aspects of the Memory Manager to manually map memory in and out of their address space. These functions are typically used when writing Modules that access hardware or manage their own memory.

Listing 11: *Memory Mapping Prototypes: (PMemoryServicesLibrary.h)*

```
// PROTOTYPES ------------------------------------------

// Mapping Functions
Ptr GetMCBase(void);
Ptr GetMCExtent(void);

uint32 GetPageSize(void);

#ifdef OS
Ptr MapPage(Ptr HWAddress);
void UnmapHWPage(Ptr HWAddress);
void UnmapPage(Ptr MappedAddress);
#endif OS
```

Note that the functions that perform hardware page mapping are not available when the Kernel is hosted on an existing operating system.

5.2.2 Kernel Memory Services: Internal API

Functions exist to create and destroy Module contexts. These are used internally by the Kernel Thread API. When threads are created a default amount of stack space is mapped into the Module Context. When a module is destroyed, any memory mapped into that Module Context is automatically unmapped.

5.3 Kernel Threading Services

The basic operating unit is a preemptive thread. Every module has one or more thread. These threads are based on a threading library, either Pthreads or the Apple-Daystar MP Library on the Mac OS.

The runtime model uses a slightly modified round robin scheduling algorithm with module priority implemented via load balancing.

As all computing processes are intrinsically bursty, the scheduler is based on message load. Modules that have handled all pending messages and do not require idle processing time are put to sleep using the thread sleep mechanism. When the dispatcher sends a message to a module, it automatically wakes the corresponding thread. As all modules are preemptively multitasked, awake threads can be scheduled in a round robin fashion. This allows an extremely simple scheduler (The scheduler built into the threading library can almost always be used) to efficiently deal with large number of processes that are not always requiring CPU cycles. This mechanism also results in a near linear slowdown as processor load increase.

While this system of scheduling is fast and simple, it does not support more advanced capabilities such as priority scheduling. A simple mechanism to implement this capability is to place higher priority modules on a less loaded CPU or system. This is done with the system level servers. As the processing power available to a module is dependent on the number of modules on that processor, the less modules running, the more time, and consequently processing power available.

No critical section code synchronization mechanisms are provided. Synchronization at a code level is not recommended, and no services for such are provided. While this lack of synchronization mechanisms may be seen as a problem, there are many Module level synchronization patterns that can be used to duplicate their capabilities. For example, when message order and time become factors, protocols such as the sliding window can enforce serialization and in order delivery.

Module Threading Functions (ThreadServicesLibrary.h)

```
// PROTOTYPES -----------------------------------------

// Thread creation and destruction prototypes
OSErr ThreadCreate(&ModuleThreadIDPtr)
OSErr ThreadDestroy(ModuleThreadIDPtr)

// Thread Utilities
void ThreadSuspendSelf(void)
OSErr ThreadSuspend(ModuleThreadIDPtr)
OSErr ThreadResume(ModuleThreadIDPtr)
Boolean isThreadRunning(ModuleThreadIDPtr)

// Placeholder function for cooperative multithreading
void ThreadYield(void)
```

Threads created using the functions listed above live within the context of the Module that created them. These threads can be used to segment tasks within a Module. For example, a server could spawn a thread for each request a specific handler gets. It should be noted that if these thread libraries are used, then the Module should be re-designed. Almost all modules do not require additional threads. If a module writer does decide to have multiple threads, they are responsible for writing the mechanisms that allow communication between threads and any necessary synchronization.

The function that is most commonly used within a Module is the

ThreadSuspendSelf call. This function suspends a thread until it receives another message. This function is called by default by the PullMessage function if there are no messages pending. This allows Modules that have no work waiting for them to be suspended allowing other Modules to use their CPU time.

5.1.3 Kernel Queuing Library

Two approaches are available to perform queuing for messages: Module based queues, and System Queues. In Module Queues, the queue and messages are stored in the memory space of the Module that is receiving the message. In System Queues, the queue and messages are stored in the system memory. Module based queues were chosen as they provide additional isolation from each other, and reduce the amount of protected code that needs to be written. It also results in a performance increase, as less mode switches are required.

Module Queue Post and Pull pusdocode

```
/*

Post Message
{
  Look up destination process information
  Verify Security privileges to post to
Module
  Allocate memory in destination process
page
  Copy message contents to allocated memory
  Add message to Module queue in
destination process
}

Pull Message
{
  Remove message from local Module queue
}

*/
```

Modules interface with the Queuing Library to post and pull messages. There are three mechanisms to send events from Modules.

The first method involves filling in all of the fields of the message structure except for the source NID, source PID and Timestamp fields. The PostMessage function can be called. Alternately, the individual values to be assigned to the values of the message fields can be passed to the PostMessage. Various overloaded functions are available to accept different numbers of parameters for the six fields that form the user defined portion of the message.

IPC Services Prototypes: (Kernel_IPC.h)

```
// PROTOTYPES  ---------------------------------------------

// Message Posting
OSErr PostMessageRaw(
  EventStructurePtr theEvent);
OSErr PostMessage(uint32 EventType,
  uint32 DestinationProcessID,
  uint32 DestinationNodeID, uint32 param1,
  uint32 param2, uint32 param3,
  uint32 param4, uint32 param5,
  uint32 param6);

// Message Pull
Boolean PullMessage(
  EventStructurePtr theEvent);

// MACROS  -------------------------------------------------
PostReply(EventStructurePtr theEvent,
  uint32 EventType, uint32 param1,
  uint32 param2, uint32 param3,
  uint32 param4, uint32 param5,
  uint32 param6);
PostReplyOne(EventStructurePtr theEvent,
  uint32 EventType, uint32 param1);
PostError(uint32 DestinationProcessID,
  uint32 DestinationNodeID,
  uint32 ErrorType);
PostErrorReply(EventStructurePtr theEvent,
  uint32 ErrorType);
```

The PostMessage routines is the only function that calls protected mode code in the message passing system. Protected mode access is required to allow access to memory mapped into different modules. When the Kernel is running in protected mode it can directly manipulate the required data structures to copy the Message data.

Module retrieval occurs fully within the address space of the Module, as the contents of the messages and the input queue are also within the Module. This

allows the retrieval functions to implement additional functions if needed such as priority message sorting

5.2 Message Dispatching

As discussed above, when messages posted from Modules they are automatically are placed into a queue corresponding to their destination. Before the messages are dispatched, they are verified for correctness and security permissions. Only if successfully verified are they are posted into the destination queue.

Modules are verified by looking up the source and destination processes in the Process Tree. If the source and the destination modules exists and are located on the local system, then the message is routed. Otherwise, if the destination is on a remote node, the message is posted into the corresponding queue for it to be routed to the correct node.

Message dispatching can occur in two different places depending on the architecture used. In one variant, the dispatcher is integrated into the PostMessage function as a protected call. In the other variant, the Dispatcher is a separate Module that handles posted messages from many different modules. Both architectures have different advantages and disadvantages. As IPC performance is critical for any message based distributed system, having the dispatcher as a separate Module is not the preferred option. The advantages of having a separate dispatcher is that it allows additional debugging information to be made available and it is a cleaner design then integrating the call directly into the Module. If performance is an issue, the dispatcher is integrated directly, and the number of searches on the process table that need to be made is cut in half.

5.2.2 Security Verification

The ADE environment contains several systems to implement basic security. As the only way that Modules can communicate is via messages, a variety of restrictions are placed on this communication vector.

Every module has a Security Clearance level. This level is inherited from the module that created it. Modules can create other modules with equal or lower security clearances, and Modules can only communicate with other modules with an equal or lower security clearance. This permits domains of modules at different levels to preserve the secrecy of information.

Modules are also assigned a type identifier. These types, System, User, etc, govern how modules can communicate between types. The current mechanism is that modules may send messages to other modules only if the type is no less then one higher or lower then the type of the sending module.

Permission Requirements:

- Must be from a valid process
- Must be destined to a valid process

For messages destined to non-system process, the Message must be sent to Modules owned by the same user and created in the same session.

Destination Module must have same clearance as the Source Module

Destination Module Process Type value must be no more then one level higher or lower then the Process Type value of the Source Module

Most of these security mechanisms are only visible when they are required. Checks three to five are largely configurable by the software developer.

Check three maintains user and process boundaries preventing programs being run by different users on the same system from interacting via messages. If two programs run by different users or during different sessions need to be able to communicate, higher-level communication mechanisms should be established. These services for session and user level IPC will be established at a later date.

Clearance Levels are largely used within the system for logical separation. As the programmer can set type values, they are useful for ensuring secured areas of the software cannot be directly accessed without going through the wrapper API.

Other checks are performed to prevent forged messages and invalid messages. These checks are performed before the message is dispatched. If a security violation is detected, a corresponding return code is returned.

5.3 Servers

A server is a Module that offers a service that forms a core portion of the operating environment. The definition of a server in the system is a system level module that provides a service to other modules.

5.3.1 The Process Server

The Process Server manages the creation and destruction of Modules. It can also be used to check if Modules are running. It

EVENT_PROCESS_CREATE This event results in a Module code being loaded from disk and instantiated in memory. This is the primary mechanism used by Modules to load other Modules. S1 --> File location identifier of Module to create Must be of Executable Type Must have user Executable flag TRUE The File Location Identifier is a String Handle of the partial path to the Module file. Modules should be placed within the ADE folder in the Modules Folder. An example string would be "\pModules:My Module".
EVENT_PROCESS_CREATE_RESULT When a Module is created, the result of the create operation is returned to the Module that sent the Create message. S1 <-- Module ID S2 <-- Create Operation Result
EVENT_PROCESS_TERMINATE This call sets the Module Terminate Flag to true. Usually, EVENT_DESTROY messages are sent to terminate a Module, but in some cases, these messages are ignored. This call is rarely used, and should be avoided. S1 --> Module ID Must have permission to terminate program
EVENT_PROCESS_TERMINATE_RESULT When a Module is terminated, the result of the create operation is returned to the Module that sent the Terminate message S1 <-- Module ID S2 <-- Termination Operation Result
EVENT_PROCESS_EXISTS Allows a Module to check if a specific Module is running. The existence of Modules is only reported if the security permissions are such that communication can occur. S1 -> Module ID
EVENT_PROCESS_EXISTS_RESULT Returns the result of the query of if a specific Module is running. S1 <- Module ID S2-4 <- Module Status

depends on the File Server to access file systems where Module code is stored on. In this development version, the File Server is incomplete, so temporary code exists in the Process Server that permits direct access to the Mac OS file system. Currently at this stage in the Mac OS version Modules are based on the Code Fragment manager. This allows ordinary development environments to be used to write Modules. For more information on exactly how Modules need to be formatted and placed to be recognized by the Process Server are covered in the read me included with the software.

Other Servers are to be added to this document as their MPI (Message Passing Interface) are finalized.

6. Module Development Patterns

6.1 Message Delivery Patterns

How messages are sent from Module to Module is critically important when designing software for the ADE system. Some of the patterns discussed below illustrate the basic message passing patterns used in the system. By combining these different patterns many different architectures can be built.

6.1.1 Many to One

Several different Modules can send messages to a single Module by opening ports to the same Module.

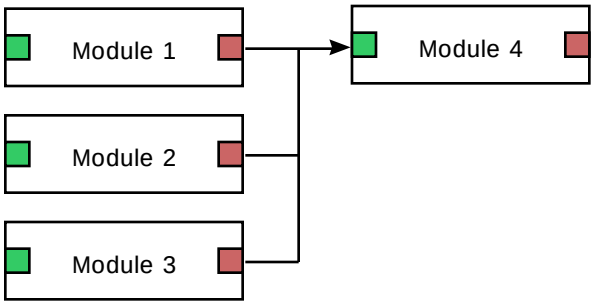

Figure 6.1: *Many to One*

The Many to One pattern is used extensively in the ADE system. Many Modules provide common functionality to many different modules simultaneously. As messages from many different modules can be received and handled from the same queue, services can be easily shared.

6.1.2 Feedback

A slight variation on the Many to One pattern is where a Module sends messages to itself. This mechanism can be used to set up feedback loops and to provide easy error handling and status monitoring.

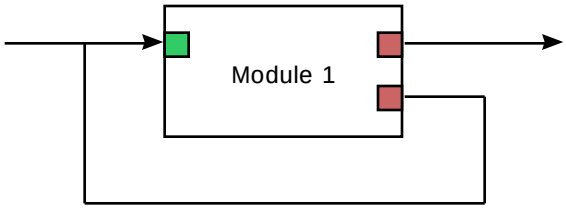

Figure 6.2: *Feedback Message Delivery*

As an example, a Module could use this pattern to keep a log of sent messages by copying all outgoing messages. A special handler for all messages with a source equal to the destination would handle the messages to log.

6.1.3 One to Many

As there are no multicast mechanisms built directly into the system, Modules must be used to provide similar functionality. Currently, Bus Modules permit one to many relationships. This is currently an area of investigation to see if multicast mechanisms with greater capabilities then offered by the bus module is required.

For example, all Modules that would like to receive EVENT_HW_MOUSE_MOVE events could contact the MouseMove Bus and subscribe. This allows modules that are interested in specific event type that is relevant to more then one module to receive it.

If modules wanted to be notified by an event when the mouse entered or left a rectangle they registered with a module, a

cross between the One to Many pattern and the Dispatcher pattern would be used. The Modules would register their rectangles, and the MouseMove module would constantly filter the stream of raw EVENT_HW_MOUSE_MOVE events into EVENT_HW_MOUSE_ENTER and EVENT_HW_MOUSE_LEAVE events that would be dispatched to the corresponding module.

6.2 Module Patterns

6.2.1 Pass to Parent

The Pass to Parent Pattern is very useful. In this pattern, Messages that are not handled by a Module are automatically sent to the Module that created it. This has the many uses in user interfaces, as the concept of expanding focus is central to today's UI.

6.2.2 Pass to Child

The Pass to Child Pattern is also extensively used in user interfaces. An example is when a window is deactivated, an EVENT_UI_OBJECT_DEACTIVATE message is sent to every object that is contained in the window.

6.3 Handler Patterns

6.3.1 Dealing with remote operation dependencies

In some instances, the order in which operations take place is critical. For example, changing the order in which matrixes are multiplied will result in an incorrect answer. This problem is usually avoided by the strictly linear nature of procedural programming languages. While this is still true in the ADE environment, there are some cases where remote processing must occur before a handler can finish. For example, in the Handler below, operation D depends on the completion of Handler C in a remote Module.

```
Handler 1
{
  operation A
  operation B
  Send message to perform operation C
  operation D
}
```

Traditionally, when operation D depends on operation C the mechanism used to ensure completion of C before D is an blocking RPC call. While this will work, it is a better approach to restructure the handler into two parts. In most cases, the resulting restructured handler will be clearer and easier to debug.

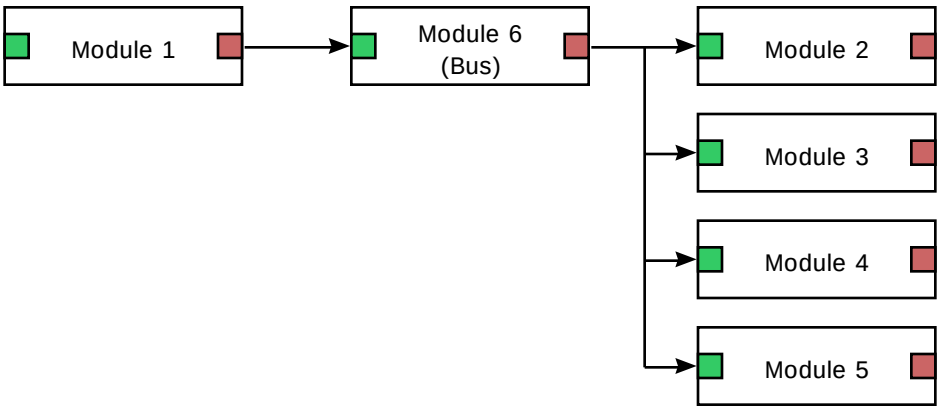

Figure 6.3: *One to Many Pattern*

```
Handler 1
{
  Operation A
  operation B
  Send message to perform operation C
}

Handler 2
{
  operation D
}
```

For this segmentation to work, the Handler that performs operation C must have the capability to return to the caller a reply message when the process has successfully or unsuccessfully completed.

This also makes error handling very elegant. If C requires a shared device that is currently in use, the device could be continuously polled by the following change to Handler 2.

```
Handler 2
{
  if(inUse)
  {
    Send message to perform operation C
  }
  else
  {
    operation D
  }
}
```

While it would be a better design to have a manager for the device that notifies interested modules when the device becomes available, this style of fragmentation makes inclusion of advanced error handling and fault tolerance easier then in traditional programming.

6.3.2 Dealing with missing messages

If a bulk data transfer is occurring, message payload can be copied into an array. When a message is received the first word of the payload indicates where it fits in the block. The remaining five words are copied, and if the location is one higher then the value of the lower counter, the lower counter is incremented. When the upper counter gets higher then the lower counter by a predefined range, messages can be sent to the sending Module to resend the missing Messages. These resent messages will be transparently merged into the array without requiring any additional coding.

6.3.3 Simulating RPCs

This is discouraged, but will be supported.

As messages destined to a Module are queued, waiting for a message within a handler will result in all messages being received being blocked. To allow a Handler to wait for a message without blocking pending messages, the HandleWait call is made available.

HandleWait will block until a message of EventType is received. To use HandleWait, the Module must be structured differently and must be Reentrant. When HandleWait is called, the Module enters a specialized loop. Messages are received, and if they do not match the check criteria, the LoopBody function is called. This permits messages to still be handled.

Module Structure using HandleWait

```
OSErr HandleWait(uint32 EventType,
  void* ModuleLoopBody);

void main(EventPtr theEvent)
}
  // Perform Initialization
  While(!TerminateFlag())
  {
    PullMessage(&theEvent);
    LoopBody(&theEvent);
  }
}

void LoopBody(EventPtr theEvent)
{
  Switch(theEvent->EventType)
  {
    case EVENT_MYMESSAGE:
    {
      // Post an Message
      HandleWait(EVENT_MYRESPONSE,
        (void*) LoopBody);
      break;
    }
  }
}
```

7. Mac OS Implementation

7.1 Mac OS Runtime Implementation

Due to limitations of the Mac OS, only one of the two system level requirements are met in the Mac OS version of the ADE environment. Preemptive threads are made possible by the Apple-Daystar Multiprocessing API, but protected memory, or more specifically, supervisor level page management, is not accessible to the programmer and is not compatible with the MP library.

7.1.1 Existing Relationships of Components

On the Mac OS, some modules run as preemptive threads, and others run as cooperative threads. As the Mac OS Toolbox offers too much functionality to resist, Modules that use toolbox calls must run as a cooperative thread. These modules then proxy the toolbox services to other modules that run preemptively. When the Carbon environment is released with Mac OS X, some of the Carbon APIs will be thread safe and thus can be called from MP tasks.

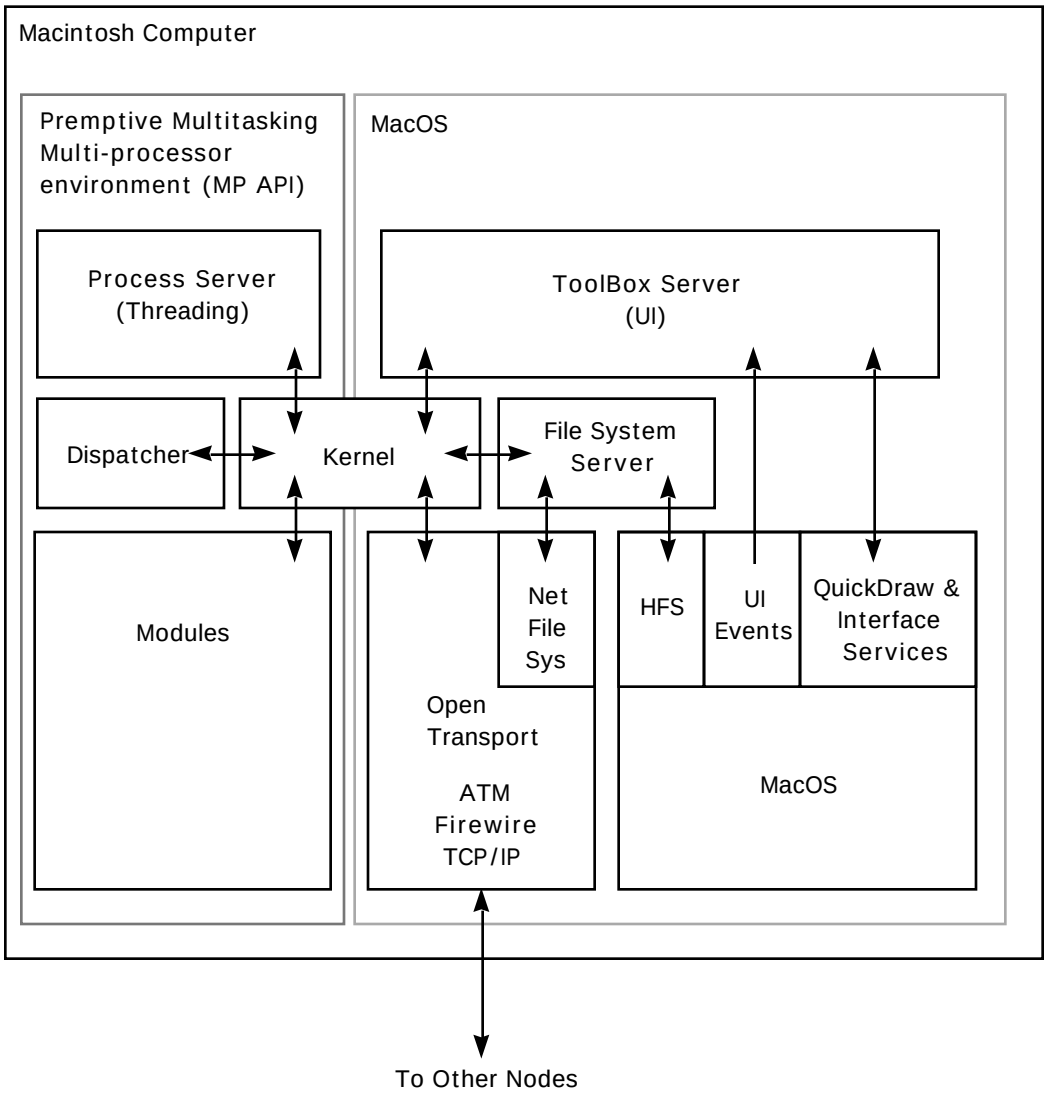

Figure 7.1: *Existing Component Relationships*

7.1.2 Current Limitations of Mac OS Implementation

While the Mac OS fulfils the primary requirements of the platform, modules are not in their own protected memory space. Thus a misbehaved modules can take down the system. While this is a concern, this weakness results in the temptation for developers to bypass mechanisms made available to perform inter-process communication and data transfer. Sharing memory directly is strongly discouraged, as it introduces complex relationships, does not port to other platforms hosting the environment and compromises the spirit of the environment. In addition, direct inter-module memory accessing will result in code breaking when hosted on multiprocessing and distributed systems.

Problems with the lack of protected memory:

- Modules can crash the system
- Modules can bypass security mechanisms
- Modules can interact in non-standard ways

The architecture is designed to avoid these three problems by making it difficult to directly access Kernel and Module structures.

7.1.3 Software included with this paper

Included with this paper is a development version of the ADE environment discussed in this paper. This release includes the environment runtime files, sample projects, a CodeWarrier project template and updated documentation about how to use the functions provided. If multiple versions of this document exist, this is document revision 0.76. The document with the larger number is the most current version.

7.1.4 Components & Icons

The Kernel is the core of the system. It manages memory, threads and process information.

ADE Libraries are core system libraries that are required to allow the ADE to run on top of an Operating System. They act as abstraction layers, insulating the Kernel and Servers from the nuances of OS specific APIs.

ADE Servers are modules that perform system domain functions for other modules.

ADE Applications take advantage of the Host Operating System to startup and provide debugging functionality to the system.

ADE Modules are the "programs" of the system and are loaded and executed by the system.

7.1.5 Other Components

The Bootstrapper

The bootstrapper is a small Macintosh application that prepares and starts up portions of the Kernel. It reserves a partition of memory, initializes internal data structures and starts up the critical servers required for operation. When the Kernel, Process Server and Dispatcher are running, it then starts the Startup Module.

The Network Router

The Network Router uses ADSP to create connections between systems running the ADE environment. As this component is still in early testing stages, it is not installed by default, and the system network topology must be configured

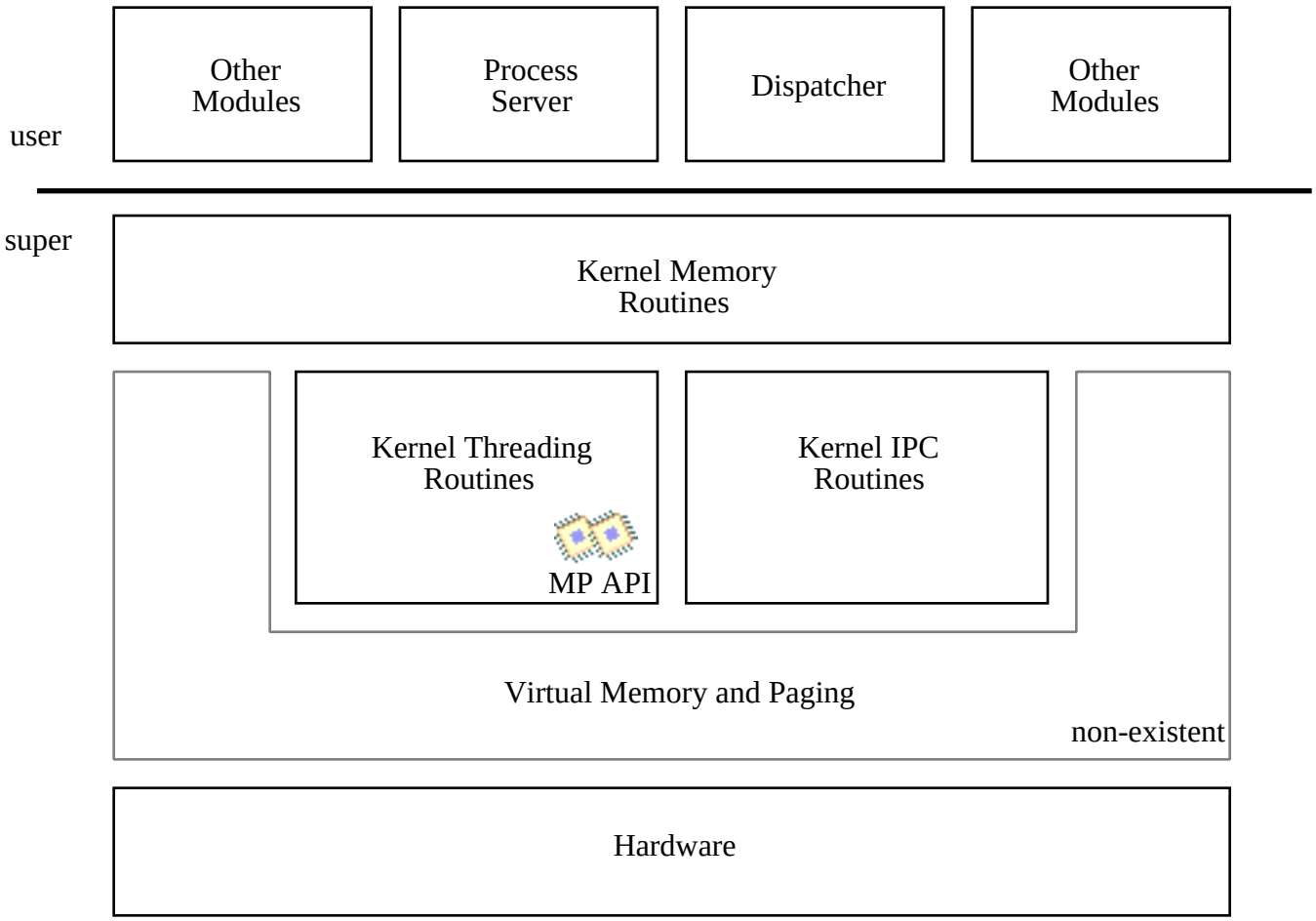

Figure 7.2: *Mac OS Implementation Diagram*

manually. Refer to the documentation with this component for more information.

7.2 Mac OS Operating Environment Implementation

The current implementation of the ADE environment on the Mac OS is outlined in the below diagram.

7.3 Libraries and External Components

The demos and SDKs provided with this paper are built in the Metrowerks development environment. You will need Metrowerks Pro 2 to build this system.

The Multiprocessing extensions must be installed if the preemptive version is to be used. The Daystar / Apple MP development SDK is included with the package. Note that CodeWarrier must be installed on your startup hard disk for debugging of preemptive Modules to be enabled.

The included DebugWindow program allows the programmer to view internal asserts, errors and messages from within the Kernel and servers. This can be invaluable for troubleshooting problems that are related to system components.

8. Future Development

The implementation of the ADE environment that is included with this paper is incomplete at best. Many improvements, both to the core system and Modules are planned over the next year.

These additions and improvements include support for additional platforms, new servers and modules and enhanced distributed capabilities.

As the ADE environment is designed to be easy to implement on top of a wide range of operating systems, ports to additional operating systems are not difficult. While access to preemptive multitasking and protected memory APIs are recommended, they are not required. Tentative ports to Unix using the PThread library have been tested. At this time, non-Mac OS implementations are largely experimental.

Many new servers will be added to the system. These include a distributed file system server, module migration across nodes, a graphical user interface server and hardware servers.

A library of reusable modules will be built up and refined to provide a robust starting point for application development. This will facilitate the development of a visual software development tool that will permit modules to be programmed together.

All of the components described in this paper will continue to be refined and improved. Documentation will be provided, and information about the system will be posted on the Web.

The most current version of this document, the SDK and examples, and additional software developed for the ADE environment can be found at:

ADE Home Page
http://web.uvic.ca/~dslik/ADE/